Proposition de recherche

Extraction du chemin critique de l'exécution d'applications distribuées par traçage noyau juste à temps

Étudiant Ph. D.
Francis Giraldeau

Directeur de recherche
Professeur Michel Dagenais
École Polytechnique de Montréal

Président du jury
Professeur Samuel Pierre
École Polytechnique de Montréal

Évaluateur externe
Professeur Samar Abdi
Université de Concordia

Département de Génie Informatique
École Polytechnique de Montréal

Hiver 2012

Table des matières

Index des tables

Index des illustrations

1 Introduction

Les systèmes infonuagiques commerciaux de grande envergure sont en utilisation croissante. La complexité de ces systèmes provient du fait qu'ils sont implémentés sur des architectures multicoeurs, qu'ils utilisent une couche de virtualisation et qu'ils sont constitués de technologies hétérogènes. L'espace de configuration de ces systèmes est exponentiel. La complexité provient aussi de l'adaptation constante du système en fonction des nouveaux besoins et de l'adaptation de la capacité. La problématique qui nous intéresse concerne l'observation des performances sur de tels systèmes, dans le but de déterminer les causes d'un problème de performance intermittent et difficile à reproduire. Les outils utilisés dans l'industrie pour le diagnostic des problèmes de performance comprennent les métriques d'utilisation des ressources, les outils de profilage, la surveillance du réseau, des outils de traçage, des débogueurs interactifs et les journaux systèmes. L'information de ces différents outils est corrélée manuellement pour déterminer la cause du problème [1]. L'observation de l'exécution de système distribué est un enjeu majeur pour en maitriser la complexité. En particulier, le chemin critique de l'exécution est essentiel pour comprendre le comportement d'une application distribuée. Ce chemin identifie les portions de l'exécution qui limitent la performance et celles qui ont un effet sur le temps de réponse.

1.1 Question de recherche

Cette proposition de recherche vise à étudier les méthodes et les algorithmes pour extraire le chemin critique d'une application distribuée à partir d'une trace noyau, tout en réduisant le surcout au minimum et en préservant les propriétés temporelles de l'exécution. Notre défi consiste à caractériser le temps écoulé d'une exécution de manière précise sans modifier le code source de l'application, que l'on considère comme une boîte noire, dans le but d'évaluer la performance d'une application et trouver des problèmes des problèmes de performance rares et difficiles à reproduire. À notre connaissance, il s'agit d'une approche nouvelle pour analyser de manière générique la performance d'applications distribuées.

La disponibilité d'outils basés sur les résultats de cette recherche améliorerait radicalement la manière dont les développeurs et analystes évaluent et trouvent les problèmes de performance sur des systèmes distribués, et en ce sens, représenterait une contribution significative.

1.2 Objectifs

L'objectif général de cette recherche est :

> Fournir des algorithmes et des outils d'analyse de traces qui permettent à des administrateurs systèmes et des programmeurs de comprendre les performances de l'ensemble de l'exécution d'une application distribuée.

Les objectifs particuliers qui découlent de l'objectif général sont :

1. Développer l'instrumentation et la sémantique pour extraire le graphe d'exécution d'une application depuis une trace noyau.
2. Extraire en ligne le chemin d'exécution d'une application distribuée.
3. Déterminer le chemin critique de l'exécution d'un traitement.
4. Calculer les ressources utilisées dans chaque composants pour une exécution.

En résumé, le système doit profiler l'exécution du système distribué dans le but de planifier les capacités, de déterminer les goulots de traitement et de surveiller le système en temps réel pour l'isolation des exécutions anormales.

Pour que le système soit applicable en pratique, les contraintes suivantes doivent être rencontrées :

1. Le système doit supporter des systèmes distribués sur une grappe d'ordinateurs multicoeurs.
2. Le système doit avoir un faible surcout pour que son déploiement sur des systèmes en production soit applicable.
3. Le système doit avoir un faible impact sur propriétés temporelles de l'exécution.

4. Le système doit être indépendant de l'architecture matérielle et transparent aux applications.
5. Le système doit produire cette analyse de manière fiable, même dans des conditions de forte charge.

L'approche boite noire est considérée importante pour s'appliquer dans le contexte d'un système aux technologies hétérogènes. Le traçage au niveau du système d'exploitation répond au besoin d'indépendance du matériel et des applications et pour cette raison, constitue l'objet de cette recherche. Nous pensons que le traçage noyau a le potentiel d'induire un surcout faible pour l'observation d'applications distribuées.

1.3 Revue de littérature

D'après le contexte et les spécifications établies, cette section décrit l'état de l'art dans ce domaine, afin de démontrer l'originalité de la contribution présentée à la section suivante.

La revue de littérature a été réalisée en utilisant les moteurs de recherche mis à la disposition par l'École Polytechnique de Montréal, entre janvier 2011 et février 2012. Les articles proviennent principalement des revues de l'IEEE, ACM et Usenix. Google Scholar a été utilisé pour compléter les recherches.

Les concepts et les mots clés utilisés pour la revue de littérature sont présentés au tableau 1. Une première requête a été réalisée en combinant ces concepts. Les articles pertinents retenus après cette première recherche ont été utilisés par compléter la recherche d'après les articles en référence, afin de s'assurer de couvrir l'ensemble des travaux du domaine. La recherche cible principalement les 10 dernières années, sans toutefois s'y limiter.

Logiciel	Distribué	Observation	Performance
operating system kernel virtual machine program source code binary executable assembly software library	transaction remote procedure distributed network protocol message passing data center high availability cloud computing client server	tracing reverse engineering observation recording benchmarking instrumentation execution graph debugging runtime veritication	critical path scaling queuing model profiling throughtput latency performance counter hardware counter

Tableau 1: Concepts et mots clés de recherche

L'inventaire comprend environ 210 articles. Nous constatons qu'il s'agit d'un domaine de recherche actif par un volume considérable de contributions récentes. Le volume de publications répertoriées selon l'année est présenté à la figure 1.

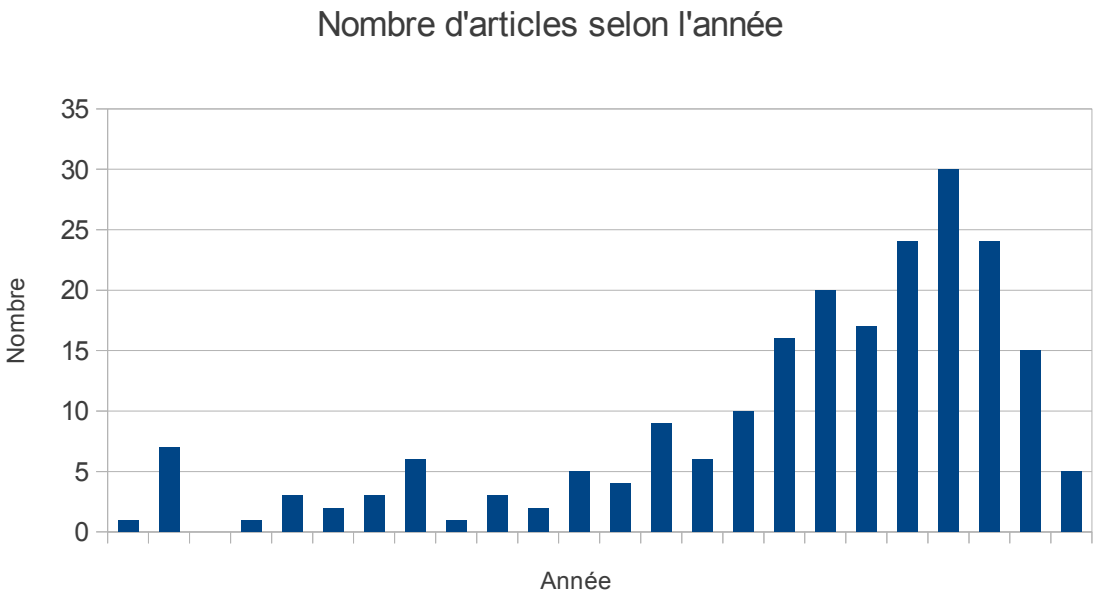


Illustration 1: Répartition dans le temps des aricles identifiés

La revue commence par la présentation des travaux antérieurs du laboratoire DORSAL. Cette section positionne la présente proposition dans le cadre des travaux du groupe de recherche. Ensuite, il est question des deux catégories principales d'analyse distribuée de systèmes en boite noire, soit les méthodes basées sur la corrélation statistique des paquets

réseau et les méthodes précises d'analyse de l'exécution distribuée [2]. Ces deux catégories sont expliquées et comparées aux sections 1.3.2 et 1.3.3 respectivement. La section 1.3.4 recense les techniques de calculer de chemin critique d'exécution, un algorithme essentiel à notre étude. La revue se termine par l'inventaire des traceurs systèmes à la section 1.3.6, de manière à comparer leurs fonctionnalités respectives et d'une synthèse à la section 1.4.

1.3.1 Travaux du laboratoire DORSAL

Le laboratoire Distributed Open Reliable Systems Analysis Lab (DORSAL) conduit un ensemble de recherche sur les systèmes distribués multicoeurs pour des applications critiques, en particulier dans le domaine du traçage. Les partenaires actuels du laboratoires sont CAE, la Défense Nationale du Canada, Efficios, Ericsson, Opal-RT et Révolution Linux. La proposition de recherche s'inscrit dans la démarche globale du laboratoire. Voici un résumé des contributions principales.

Le laboratoire a démarré le projet Linux Tracing Toolkit (LTT) en 1999 [3], un traceur noyau pour le système d'exploitation Linux. Son successeur LTT Next Generation (LTTng) offre une meilleure mise à l'échelle pour des architectures multicoeurs par l'utilisation de structures de données sans verrous et des optimisations poussées [4–6]. LTTng a été étendu pour supporter le traçage haute performance en mode utilisateur [7]. Des travaux portant sur la synchronisation temporelle de traces distribuées ont été réalisés [8], [9]. La version de LTTng 2.0, publiée en mars 2012, représente l'état de l'art dans le domaine du traçage. Cette version supporte le traçage en mode noyau, en mode utilisateur, les points de trace dynamiques, les compteurs de performance. Cette infrastructure de traçage utilise des tampons par processeurs sans verrou sophistiqués et enregistre au format Common Trace Format (CTF), un format standard adapté au traçage.

Du point de vue de l'analyse des traces se trouvent deux outils de visualisation, soit LTT Viewer (LTTV) et le plugin LTTng pour Eclipse. Ces deux outils fournissent un graphique de l'état des processus selon le temps, un histogramme et des statistiques de base [10]. Des progrès ont été réalisés dans l'utilisation de traces pour des analyses de sécurité [11], la reconnaissance de séquence d'évènements à l'aide de machines à états finie [12], l'étude des

relations de blocage entre les processus [13] et la rétro-ingénierie [14]. Un index hiérarchique d'intervalle a été développé, ce qui rend possible de retrouver l'état du système à n'importe quel moment de la trace en temps logarithmique [15].

1.3.2 Méthodes de corrélation statistique des paquets réseaux

Les méthodes statistiques retrouvent des liens entre les évènements par une technique de corrélation temporelle [16–18]. Les évènements en entrée et en sortie sont observés et le temps écoulé est enregistré. Avec un échantillon suffisant d'évènements, l'algorithme rapporte la probabilité du lien causal.

Constellation [19] se base sur une technique statistique pour déterminer un réseau de dépendance entre des ordinateurs. Le trafic réseau est enregistré passivement. Les adresses et les ports de communication sont utilisés pour l'analyse. L'hypothèse est faite qu'une requête est probablement la cause d'une autre si elles sont rapprochées dans le temps. En enregistrant un profil sur une longue période, un test statistique est utilisé pour discerner les requêtes reliées. Le taux de faux positif obtenu est de l'ordre de 2% pour les requêtes HTTP. Une approche similaire a été proposée par Aguilera et al. [20].

Les méthodes basées sur l'observation passive du réseau présentent l'avantage de ne pas ajouter de latence supplémentaire lors de l'exécution, car l'instrumentation n'est pas sur le chemin critique. Aussi, cette approche s'applique pour des applications quelconques. Les désavantages sont la présence de faux positifs ou de relations qui ne sont pas identifiées. En particulier, une faible précision est obtenue pour les chemins de communication rares. Enfin, l'observation unique du réseau ne permet pas de déterminer les liens entre les processus spécifiques impliqués, ni les liens entre les processus locaux d'un ordinateur, à moins d'avoir une instrumentation supplémentaire.

1.3.3 Méthodes précises d'analyse de l'exécution distribuée

L'approche précise nécessite la connaissance de la sémantique des évènements pour mettre à jour un modèle du système. Le modèle et les évènements doivent correspondre. L'emplacement de l'instrumentation change la nature des évènements disponibles. Les

méthodes précises présente une difficulté à l'égard de la mise à l'échelle, car la quantité d'évènements à traiter croit de manière proportionnelle à la taille du système observé, contrairement aux techniques imprécises qui font un échantillonnage.

L'instrumentation de l'application ou des librairies est une manière simple et efficace pour instrumenter une application distribuée de manière précise [21–32]. L'instrumentation statique du programme nécessite la modification du code source et ajoute une dépendance sur une librairie de traçage. Il existe des méthodes d'instrumentation dynamique modifiant sur place le programme exécuté [33], [34], mais les techniques varient en fonction du langage de programmation et de l'environnement d'exécution. L'instrumentation d'une librairie a l'avantage d'éviter la modification du code source, mais restreint l'analyse aux applications utilisant cette librairie, ce qui ne permet pas d'observer une application quelconque. L'instrumentation au niveau applicatif n'a pas accès aux évènements du système d'exploitation, comme les interruptions et l'ordonnancement, ce qui limite la possibilité de caractériser le temps écoulé. Pour la suite de cette section, les approches les plus intéressantes et pertinentes au sujet sont détaillées.

Google Dapper [21] se base sur l'instrumentation de la librairie de communication RPC pour l'analyse des requêtes distribuées. La librairie assigne un identifiant unique aux requêtes au point d'entrée du système. Cet identifiant sert ensuite à corréler tous les évènements relatifs à une requête. Chaque requête RPC est tracée récursivement pour constituer l'arbre des requêtes selon les composants impliqués. Pour minimiser le surcout et activer le système en production, les requêtes sont échantillonnées, ce qui permet de mettre à l'échelle l'observation. Une requête est tracée complètement ou pas du tout. Le temps passé dans un composant est décomposé entre le temps de communication, d'attente et de traitement. Les requêtes sont agrégées par similarité pour calculer la variance et détecter des anomalies. Cette méthode ne nécessite pas de modification de l'application elle-même, mais ne s'applique que pour les applications utilisant la librairie instrumentée, ce qui limite sa généralité. D'ailleurs, le modèle de performance utilisé est réducteur et spécifique aux applications RPC utilisées dans l'environnement contrôlé de Google. Finalement, les

auteurs se demandent eux-même comment associer l'instrumentation du noyau dans leur modèle.

Jumpshot [22] est un outil de visualisation de trace d'une application MPI. Le programme est instrumenté en le liant avec la librairie *libmpe*. Cette librairie détourne les appels aux fonctions MPI, génère des évènements reliés aux communications des programmes, puis effectue l'appel réel de la fonction interceptée. La vue de la trace montre l'ensemble des processus distribués impliqués dans le traitement. La vue temporelle montre le temps passé en calcul et en communication. Pour chaque communication, une ligne montre le lien entre l'émission et la réception d'un message. L'utilisation de Jumpshot requiert de refaire l'édition des liens de l'application, mais ne nécessite pas la modification du code source. Seules les applications MPI peuvent être instrumentées de la sorte, ce qui réduit la portée de cette instrumentation au même titre que Google Dapper. D'autre part, le traitement fait par l'application en dehors des appels à la librairie de communication MPI n'est pas visible, ce qui limite la capacité de caractériser le temps écoulé.

Stardust [27] est capable de déterminer les ressources utilisées (temps processeur, accès disque, transfert réseau et utilisation de la mémoire) pour le traitement d'une requête distribuée, ainsi que la latence induite par chaque composant. En particulier, le système produit une analyse très précise en imputant les écritures sur disque effectuées en arrière-plan aux processus qui en sont la cause. Les évènements sont reliés par la relation transitive d'un identifiant propagé aux sous-systèmes. Le système a été évalué par l'instrumentation du serveur NFS Ursa Minor. Environ 200 points de trace ont été ajoutés manuellement dans le code des composants du système pour obtenir le résultat, ce qui n'est pas appropriée pour l'observation de type boîte noire.

Magpie [29], [30] extrait le chemin d'exécution d'une requête distribué. Le traceur Event Tracing for Windows (ETW) est utilisé pour obtenir les évènements du système d'exploitation, des applications et du réseau. L'analyse de la trace est généralisée de deux manières. Premièrement, l'utilisation d'un schéma décrit la sémantique des évènements disponibles. Deuxièmement, la jointure temporelle décrit les liaisons à créer entre les évènements pour reconstituer la requête. La jointure exploite le lien transitif existant entre

les champs des évènements. La propriété de clôture transitive forme la requête complète. La requête obtenue comprend les accès disques, l'ordonnancement et les autres évènements du système d'exploitation. Les chemins d'exécutions similaires sont regroupés avec un algorithme calculant la distance de Levenshtein. Le désavantage majeur de Magpie concerne la nécessité de l'instrumentation requise du cadre applicatif avec ETW. Cette approche nécessite l'accès au code source du cadre applicatif et sa modification, ce qui n'est pas envisageable dans le cas d'une infrastructure hétérogène. Aussi, Magpie a été évalué dans le cadre de l'analyse d'un service Web, ce qui est un sous-ensemble des applications que nous souhaitons observer. Une approche similaire a été proposée par Mysore et al. [35].

X-Trace [36] retrouve le chemin causal de communication en instrumentant la couche réseau avec un identifiant. Cet identifiant est relayé au prochain composant ou propagé aux sous-requêtes de manière récursive. L'identifiant est répliqué à travers les couches réseau, doit à travers les couches applicatives, TCP et IP. Les champs d'options de TCP et IP sont utilisés pour conserver les métadonnées. La manière de propager les métadonnées à la couche applicative dépend du protocole. La trace obtenue permet de relier le chemin causal complet de manière précise à travers les zones réseau. Cette technique requiert une modification invasive des applications de manière à déterminer le type de propagation de l'identifiant à effectuer, il ne s'agit pas à proprement dit d'une méthode d'instrumentation de type boite noire. D'autre part, aucune autre information de profilage n'est disponible pour analyser la performance de l'application.

BorderPatrol [16] retrouve le chemin d'exécution d'une application distribuée. Les appels à la librairie standard C sont interceptés par le préchargement d'une librairie dynamique. Des évènements sont enregistrés lors de l'établissement d'une connexion et lors de la transmission de messages. Ce qui distingue cette étude est l'utilisation de processeurs de protocoles standard pour observer et identifier la nature des messages échangés entre les composants. L'exactitude du chemin d'exécution retrouvé repose sur trois hypothèses concernant le comportement des applications observées, soit l'honnêteté de l'application (absence de bogue et comportement non malicieux), le traitement immédiat (sans multiplexage) et le traitement identique de requêtes reçues simultanément et

séquentiellement. L'application n'a pas besoin d'être modifiée pour être instrumentée, ce qui constitue un avantage. Les processeurs de protocoles fournissent un contexte pour l'identification d'une requête. Or, l'algorithme d'analyse de la trace par une copie dans une base de données SQL souffre d'un problème de mise à l'échelle. L'analyse se fait hors ligne. Aussi, le format fixe des évènements restreint la portée de l'instrumentation aux appels systèmes et aux adresses de communications et ne permet pas de caractériser précisément le temps de traitement.

vPath [37] extrait le chemin d'exécution d'une requête par l'enregistrement d'évènement lors de l'envoi et de la réception de message de chaque fil d'exécution. Les évènements sont enregistrés par l'hyperviseur Xen, ce qui ne nécessite aucune modification au code source des applications, ni l'inspection du contenu des messages. L'implémentation fonctionne pour l'architecture x86. Chaque fils d'exécution est identifié uniquement par le tuple formé de l'identifiant du domaine et les valeurs des registres CR3 et EBP. L'analyse fonctionne pour des applications respectant deux idiomes de programmation, soit le traitement synchrone des requêtes au-dessus d'une couche de communication fiable et la répartition du travail par fil d'exécution. L'étude démontre que la plupart des applications et des intergiciels courants respectent ces idiomes. La causalité du point de vue des fils d'exécution est capturée par l'observation des évènements d'ordonnancement du système d'exploitation, tandis que la causalité entre les composants est déterminée par les messages échangés. Le modèle de traitement par un ensemble de fils d'exécution de travail (*worker thread*) satisfait l'hypothèse du comportement, et ce modèle est prévalent. Le système ne supporte pas certains modèles d'exécution en pipeline à cause de l'absence de l'instrumentation de la mémoire partagée.

Whodunit [38] extrait l'exécution d'une transaction distribuée et produit un profil de l'exécution selon le composant. Le profilage de chaque composant est obtenu avec csprof [39]. L'instrumentation des messages est effectuée pour déterminer les liens entre les processus distribués et est implémentée par la surcharge de la librairie C. D'autre part, dans le but de détecter les liens entre des processus locaux, la communication en mémoire partagée est tracée. Ceci est accompli en interceptant les appels aux fonctions de verrou,

puis en exécutant le code de la section protégée avec l'émulateur QEMU. Les adresses utilisées sont enregistrées pour identifier les processus producteur et consommateur. Whodunit est spécifiquement conçu pour les application multitiers.

PreciseTracer [2] extrait le chemin d'une requête distribué d'une requête de manière précise et en ligne. La construction du graphe d'activité de la requête se base sur les échanges de messages entre les applications. Chaque évènement enregistré associe un message TCP au processus impliqué. Ces évènements sont ensuite reliés entre eux pour former le graphe d'activité. Les graphes similaires sont superposés dans le but de présenter une vue synthèse de l'exécution. Deux approches sont utilisées pour améliorer la mise à l'échelle de l'analyse. Le traçage est activé globalement pour une courte période, ce qui diminue l'impact par rapport à si le traçage était toujours actif. Ceci diminue la charge moyenne sur une longue période, mais ne diminue pas le surcout lorsque l'instrumentation est activée, ce qui a un impact sur les requêtes traitées pendant ce temps. L'autre approche consiste à effectuer un échantillonnage aléatoire des évènements. Pour un taux d'échantillonnage de 90% des évènements du banc d'essai RUBiS, 90% des graphes sont exacts. La méthode a l'avantage de montrer que l'algorithme est tolérant à un certain taux de pertes, mais un taux d'échantillonnage de 90% ne diminue que marginalement le surcout du traçage et l'algorithme n'a pas été testé avec un taux d'échantillonnage plus faible.

Self-Propelled Instrumentation [40] instrumente dynamiquement une application distribuée en suivant le flot de contrôle de son exécution. Une librairie contenant l'instrumentation est chargée et l'exécutable est modifié sur place pour insérer un saut vers l'instrumentation correspondante. La librairie contient des fonctions de rappel définies par l'utilisateur. L'instrumentation est propagée aux processus enfants en interceptant les appels aux fonctions de création de processus de la librairie C. Lorsque l'application se connecte à un serveur, alors celui-ci est instrumenté en arrière-plan lors de l'établissement de la connexion avant de retourner. L'opération s'effectue sur l'ordinateur pair par SSH. Nous avons mesuré que l'établissement de la connexion SSH locale sur un Intel core i5 à 2.5Ghz et le serveur OpenSSH avec l'authentification par clé publique de 2048 bits à elle seule est de l'ordre de 250ms. Par conséquent, cette technique ne préserve pas les propriétés temporelles de

l'application. Aussi, l'implémentation de la propagation de l'instrumentation comporte des implications au niveau de la sécurité et de la configuration du système. Le client doit avoir le privilège de modifier l'exécutable du serveur, ce qui n'est clairement pas désirable dans un environnement commercial. Aussi, le client et le serveur doivent s'exécuter sur des ordinateurs de même architecture, ce qui ajoute une contrainte sur le matériel supporté. Finalement, l'instrumentation de binaire ne fonctionne que pour les programmes compilés et exclu les exécutables intermédiaires comme le bytecode Java.

1.3.4 Calcul du chemin critique

Miller et al. [41] présentent un algorithme pour le calcul du chemin critique d'une application distribuée basée sur les messages échangés. Le graphe d'activité d'un programme (Program Activity Graph, PAG) est défini comme un graphe dirigé acyclique, pour lequel les nœuds sont les évènements délimitant le début et la fin d'une activité et dont les arcs représentent le temps processeur utilisé. Le chemin le plus long correspond au chemin critique. Sa longueur est la somme du poids de ses segments. Leur étude compare un algorithme centralisé à un algorithme distribué pour le calcul du chemin critique hors ligne. L'accent est donc sur l'accélération du calcul hors ligne du calcul du chemin critique par un algorithme parallèle.

Hollingsworth et al. [42] proposent une méthode pour le calcul en ligne du chemin critique d'un graphe d'activité. La technique présentée évite de construire le graphe complet et de stocker les évènements. La technique consiste à adjoindre la valeur courante du chemin critique à un message envoyé. Lors de la réception d'un tel message, cette valeur est copiée dans une variable locale au processus. Le temps de traitement du processus est ajouté. La valeur à jour est transmise avec la réponse. Lors de la réception d'un message, la valeur la plus élevée entre celle locale et celle reçue est conservée.

Le calcul du chemin critique proposé par Miller et Hollingsworth est efficace pour les applications dont la performance est limitée par le temps processeur. Or, ce type d'analyse est insuffisante pour déterminer le chemin critique d'un système qui serait borné par les entrées sorties. Si le temps d'attente et de communication est inclus au poids des arcs, alors

tous les chemins du graphe ont le même poids, ce qui rend les algorithmes de calcul du chemin critique existants inopérants. Aussi, les dépendances entre les processus locaux utilisant d'autres mécanismes de communication, comme les tubes et les signaux, ne sont pas supportés, ce qui restreint le nombre de programmes qui peuvent être observés efficacement.

LTTV est un visualiseur de trace noyau développé au laboratoire DORSAL. Il contient plusieurs modules, dont un histogramme du nombre d'évènements selon le temps, une vue des ressources du système et un module d'affichage de l'état des processus. L'analyseur reconstitue l'état du système depuis la trace. En ce qui concerne l'analyse du chemin critique, le module d'analyse de dépendance de LTTV [13] utilise le traçage noyau pour déterminer la cause de l'attente dans un programme. L'analyse porte sur les blocages se produisant dans lors d'appel système. L'émetteur de l'évènement de réveil indique la cause de l'attente. L'algorithme utilisé est hors ligne et assume une structure d'attente en arbre qui n'est pas adéquate pour le cas général, ce qui est expliqué à la section 2.1.

1.3.5 Détection de goulots d'étranglement

Saidi et al. [43] présentent une méthode d'extraction du chemin critique de l'exécution pour en déterminer les goulots, qu'ils soient logiciels ou matériels. La méthode consiste à modéliser sous forme d'automate les composants à analyser, dont les évènements sont les transitions. Les liens entre les automates sont définis par l'utilisateur et forment le flot de contrôle et d'attente. Les états des automates correspondent aux arcs du graphe de dépendance. Ceux-ci sont annotés avec la durée pendant laquelle l'état est maintenu. Le système de mesure a été implanté avec un simulateur matériel. L'analyse à bas niveau du matériel se fait en instrumentant le simulateur avec des fonctions de rappel. L'analyse logicielle utilise la capacité du simulateur d'instrumenter l'entrée et la sortie des fonctions. Les automates sont annotés avec les étiquettes des fonctions. Le chemin critique est calculé sans reconstruire le graphe complet à la manière décrite par Hollingsworth et al. La méthode de visualisation du graphe proposée consiste à regrouper les graphes isomorphes et annoter les arcs par le nombre d'exécutions, le temps total écoulé et la proportion du temps sur le chemin critique. L'analyse vise spécifiquement la détection des goulots dans le

code source et le matériel, tandis que nous nous intéressons au cas général du graphe d'exécution au niveau du système. En outre, l'analyse requiert la connaissance du code source des applications, du noyau et un modèle du matériel, ce qui n'est pas envisageable pour l'analyse de type boîte noire.

L'analyse d'un réseau de files d'attente en couche (Layered Queuing Network, LQN) a été démontré efficace pour déterminer la capacité et les goulots d'un système distribué [44]. En particulier a été appliquée avec succès pour déterminer les goulots d'un système temps-réel souple de téléphonie IP [45]. Il s'agit d'une généralisation d'un modèle de la théorie des files d'attentes pour des systèmes distribués. Le temps d'attente moyen en file et la longueur moyenne de la queue sont des exemples de mesures de performance que l'analyse peut produire. Pour certains modèles simples, il existe une solution analytique. Dans les autres cas, les résultats peuvent être obtenus par simulation discrète du modèle. La difficulté reliée à l'utilisation de LQN concerne la génération du modèle de performance depuis le système. L'automatisation de cette approche est proposée pour faciliter la construction du modèle [46].

L'analyse du comportement statistique des paquets réseaux a été utilisé pour déterminer l'atteinte d'un goulot d'étranglement, sans pour autant connaître à l'avance la capacité maximale du système [47]. La distribution de la latence de communication est compilé pour une fenêtre de temps. D'après cette distribution, le facteur de dissymétrie de la distribution est calculée. Une dissymétrie positive indique que la distribution est compressée à la limite du système et donc l'atteinte d'un goulot.

1.3.6 Traceurs

Le traceur LTTng a été présenté dans la section 1.3.1. Cette section présente les autres traceurs disponibles et leurs particularités.

Event Tracing for Windows (ETW) [48] est le système de traçage développé par Microsoft pour Windows®. L'infrastructure trace autant le noyau que les applications en espace utilisateur. Des tampons par CPU sont utilisés pour diminuer le surcout lié à la synchronisation sur des architectures multicoeurs. L'outil Windows Performance Analyzer

(WPA) affiche des graphiques de l'utilisation des ressources du système, calculés depuis les évènements de la trace. L'analyseur utilise les symboles de débogages des exécutables tracés pour relier les métriques au code source des applications.

DTrace [49] est un traceur disponible pour Solaris, Mach et FreeBSD. Ses fonctionnalités comprennent le traçage du noyau des programmes en espace utilisateur, la définition d'une fonction de rappel par type d'évènement, le filtrage et le traçage spéculatif. Le traçage est défini par un script en langage D, dans lequel sont spécifiés les points de trace à activer et leur traitement.

SystemTap [50] est un traceur pour Linux. Il permet d'accéder aux points de trace au niveau du noyau, de l'espace utilisateur et aux compteurs de performance. Un script STP définit les points de trace à activer et la manière d'agréger les données en vue de leur affichage. La fonction d'affichage est appelée périodiquement, ce qui constitue la seule manière pour consulter les données. Le script STP est compilé dans un module noyau, puis chargé pour démarrer l'analyse.

Perf [51] est un traceur regroupant les compteurs de performance matériels et logiciels du noyau Linux. Il est utilisé pour profiler l'exécution du noyau ou d'un programme en espace utilisateur. Les compteurs de performance disponibles varient selon l'architecture, comme le nombre d'instructions par cycle d'horloge et le nombre d'accès invalides à la cache. Les compteurs logiciels incluent par exemple les fautes de page mineures et majeures. Les données peuvent être sauvegardées ou affichées en temps réel.

Ftrace [51] permet d'obtenir le détail de l'appel des fonctions du noyau et le temps passé dans chacune d'elle. Il existe des modules d'analyse spécialisés, comme celui pour mesurer la latence de l'ordonnanceur pour des tâches temps réel.

LTTng sera utilisé pour obtenir les résultats de cette recherche. Ceci se justifie parce qu'il offre un niveau de fonctionnalité supérieur aux autres traceurs disponibles sous Linux et que nous avons accès au code source pour effectuer nos expérimentations. Du point de vue de la performance, des bancs d'essais démontrent que le traceur noyau LTTng requiert 119 ns pour enregistrer un événement en mémoire sur un processeur Intel Xeon cadencé à

2 GHz, dans des conditions de cache optimale [4]. Nous pensons que ces performances sont adéquates pour la réalisation du projet.

1.4 Synthèse

Les approches basées sur l'instrumentation du cadre applicatif et des applications permettent d'obtenir une analyse fine du comportement du système, directement en lien avec la nature de l'application. L'avantage concerne la nature des évènements qui est directement en lien avec le domaine de l'application. Cette relation est très importante pour déterminer des correctifs ou des optimisations. Nous cherchons à obtenir le même bénéfice sans nécessiter l'instrumentation du cadre applicatif, des librairies ou des applications.

De tous les articles de la revue de littérature, l'étude de vPath est celle qui se rapporte le plus à nos objectifs. L'approche de type boite noire au niveau de l'hyperviseur répond aux critères d'une analyse indépendante de l'application. Cette étude confirme la faisabilité de cette méthode pour un système transactionnel multitiers. Il existe d'autres catégories d'applications pour lesquels le modèle de concurrence n'a pas été étudié, notamment les communications asynchrones. Ce type de communication est commun pour accroitre le parallélisme de l'exécution. Or, vPath, tout comme le module d'analyse de dépendance LTTV, fait l'hypothèse que l'attente entre les composants est imbriquée, ce qui est trop restrictif dans le cas général.

Contrairement à l'instrumentation des applications, vPath ne permet pas de faire la correspondance entre la trace et le code source de l'application. En ce sens, l'intégration du profileur statistique csprof dans Whodunit constitue une avenue intéressante pour faire ce lien, à cause de sa précision et de son faible impact de performance.

Nous n'avons pas été en mesure de trouver dans la littérature une méthode générale permettant de déterminer en ligne le chemin critique d'une application au niveau système avec une approche de type boite noire. La proposition de recherche porte sur la manière d'extraire un tel chemin critique depuis une trace noyau et étudier comment améliorer la mise à l'échelle de cette analyse tout en préservant sa précision et sa généralité. Nous souhaitons continuer le travail dans la direction du module d'analyse de dépendance de

LTTV au niveau noyau. Par rapport aux travaux antérieurs, le contexte technologique a évolué. L'infrastructure de traçage LTTng utilise maintenant des structures de données sans verrous Read-Copy-Update (RCU), ce qui diminue la contention sur les structures partagées. Nous disposons d'un index d'intervalle sur disque permettant de reconstituer l'état à n'importe quel moment. Une requête sur cet index a une complexité logarithmique et présente une utilisation de mémoire constante. Nous désirons exploiter toute l'information mise à la disposition, comme les évènements de requêtes du disque, la gestion de la mémoire, l'ordonnancement et les interruptions, pour caractériser précisément le temps d'exécution d'une application au niveau système, qui ne serait pas possible autrement. Tous ces éléments contribuent à la nouveauté du projet.

Les contributions d'ordre technologiques et algorithmiques prévues sont détaillées à la section suivante.

2 Méthodologie

2.1 Analyse préliminaire

Notre but est de reconstruire un graphe dirigé acyclique (Directed Acyclic Graph, DAG) de l'exécution d'une application. Les nœuds représentent le changement d'état de l'exécution et les arcs sont les liens entre les états. Le graphe d'exécution d'un processus dont l'exécution n'a aucun lien avec d'autre processus dégénère en une liste. Le graphe sert à calculer le chemin critique de l'exécution.

La technique d'annotation des arcs par le temps processeur proposée par Hollingsworth permet de retrouver le chemin critique d'une application limitée par le temps de calcul. Cette analyse est adaptée pour les applications de calcul scientifique, mais ne rapporte pas le bon chemin critique pour des applications dans laquelle il existe de l'attente. Pour rendre l'analyse plus générale, nous proposons d'étudier deux types de primitives, soit les évènements qui causent une divergence et une jointure dans le graphe. Ces évènements sont plus généraux, et comprennent la création et la terminaison des processus, les temporiseurs, les verrous et les entrées sorties. Un exemple d'un tel graphe est représenté à la figure 2. Il

est montré le processus A en exécution qui fait un appel à clone et crée le processus B, puis se met en attente de ce processus. L'appel à clone est traité comme un évènement divergent, pour lequel deux arcs sortants sont définis. Lorsque le processus B se termine, un arc relie la fin du processus au message de synchronisation dans le processus A. Par cette approche, on voit que peu importe si le processus B utilise ou non du temps processeur, la durée d'exécution du processus A dépend de la rapidité de l'exécution de B. Par ce fait, on peut conclure que le processus B est sur le chemin critique de l'exécution du processus A.

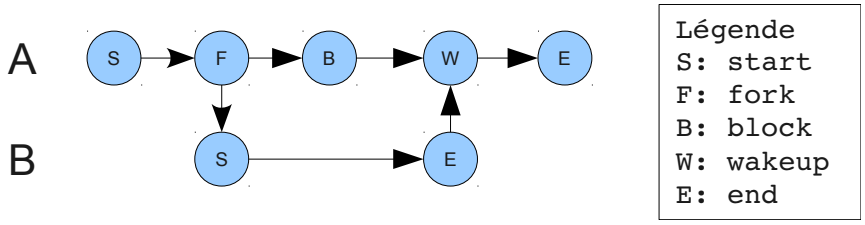


Illustration 2: Graph d'exécution de clone et wait

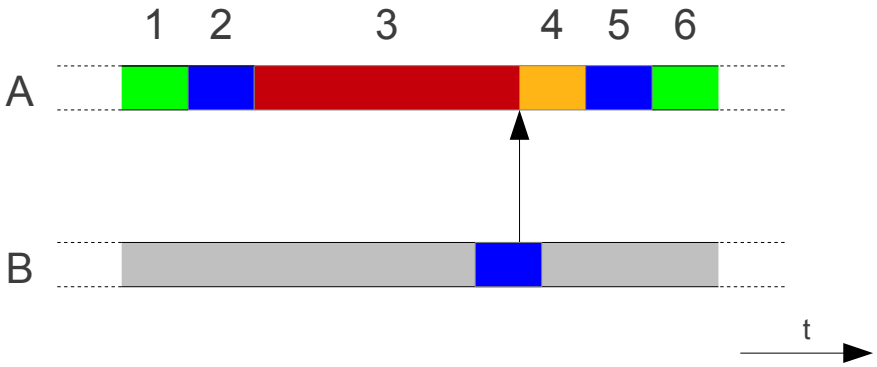


Illustration 3: Relation d'attente entre deux processus

Le principe de fonctionnement de l'analyse proposée s'appuie sur l'état bloqué d'un processus et son signal de réveil [13]. Un exemple de cette séquence est montré à la figure 3. L'état des processus A et B dans le temps est représenté, dans lequel le processus A attend la terminaison du processus B. Le processus A s'exécute en espace utilisateur (1) puis exécute l'appel système waitpid (2). Si le processus B n'a pas encore quitté, alors le processus passera à l'état bloqué (3), ce qui se traduit par un changement de contexte. Lorsque le processus B quitte, alors le système d'exploitation remet le processus A dans la file de l'ordonnanceur (4). Lorsque le processus recommence à s'exécuter, l'appel système retourne (5), puis l'application continue son traitement (6). La durée du blocage dépend du

temps d'exécution du processus B. On peut conclure qu'un segment de l'exécution du processus B est sur le chemin critique du processus A. La provenance du signal de réveil indique la cause de l'attente. Le calcul du chemin critique se fait en remontant la chaine d'attente.

Comme analyse préliminaire, nous avons étudié les blocages des appels systèmes principaux pour évaluer leurs comportements et les limitations potentielles.

- nanosleep : le signal de réveil est émis lors de l'expiration du temporiseur.
- read : le signal de réveil est émis dans un softirq indiquant que les données sont disponibles.
- select : le signal de réveil est émis si un ou plusieurs descripteurs sont prêts, et les descripteurs en cause sont identifiés.
- waitpid : le signal de réveil est émis par l'enfant lorsque celui-ci se termine.
- futex : le signal de réveil est émis par le détenteur du verrou lorsqu'il est libéré.

Pour tous les appels systèmes bloquants étudiés jusqu'à maintenant, leur comportement de blocage est le même, et nous avons été en mesure de déterminer la cause de premier degré. L'évènement de réveil est une indication du changement du flot de contrôle, qui est transféré au processus réveillé.

La cause racine du réveil est directe dans le cas de l'attente d'un processus avec waitpid. Cependant, les réveils effectués depuis un softirq sont indirects. Le défi est de remonter la chaine de causalité qui a mené à l'exécution du softirq. Dans la majorité des cas, il s'agit d'un IRQ. Son numéro indique le périphérique en cause dans l'attente. Par exemple, dans le cas d'un évènement lié au clavier, l'attente serait attribuée à l'utilisateur. Si l'interruption est émise par le contrôleur du disque, alors il doit être possible de remonter encore la chaine pour déterminer le début de la requête au disque.

Nous avons testé l'analyse de dépendance produite par l'algorithme d'analyse de dépendance de LTTV. Le scénario est celui de l'attente entre 3 processus. Le processus master effectue un appel système à clone, ce qui a pour effet de démarrer le processus child 1. Le processus

master continue son exécution, puis effectue l'appel système wait sur l'enfant en exécution. Le processus child 1 crée à son tour l'enfant child 2 et attend sa terminaison. L'exécution et le chemin de dépendance retrouvé par le module LTTV est montré à la figure 4 a). Ce chemin est incomplet, car si le temps d'exécution de child 2 est raccourci, alors le temps total d'exécution de master serait réduit. Le chemin critique passe donc par child 2, comme montré en b). Ce chemin incomplet retrouvé s'explique par le fait que l'algorithme fait l'hypothèse d'une hiérarchie d'attente imbriquée, ce qui n'est pas vrai dans toutes les circonstances. L'utilisation d'un graphe est donc nécessaire pour représenter la chaine de dépendance dans le cas général.

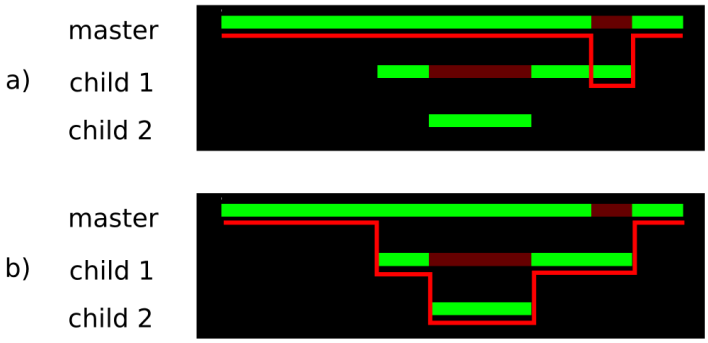


Illustration 4: Chemin critique du processus master

Nous avons développé un algorithme d'extraction du chemin d'exécution hors ligne et l'avons appliqué à la trace de l'exemple précédent. Le graphe résultant est présenté à la figure 5. On y voit les 3 processus avec les liens entre les états. Dans ce graphe, tous les chemins ont la même durée d'exécution, donc les algorithmes du calcul du plus long chemin ne s'appliquent pas. Pour calculer le chemin critique, le graphe est parcouru à rebours depuis la fin. Lorsqu'un évènement de réveil est rencontré, qui correspond à une jointure, ce chemin est suivi. Puis, le premier nœud de divergence rencontré est suivi. Cet algorithme se termine lorsque le premier état du graphe est rencontré. Le chemin visité correspond au chemin critique, et pour l'exemple correspond à SCSCSEWEWE.

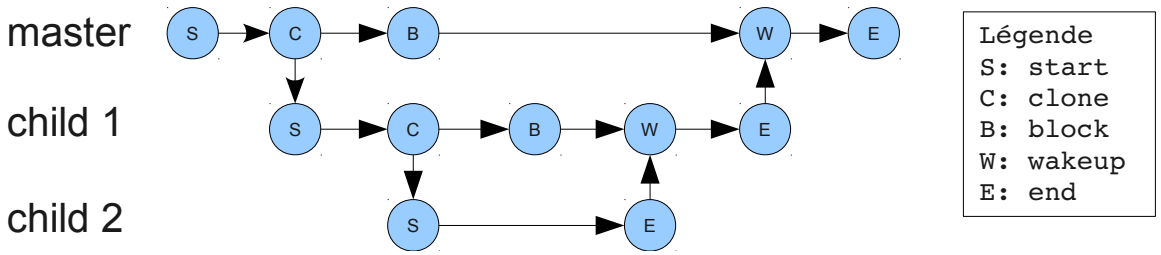


Illustration 5: Graphe d'exécution du processus master

Lors de l'analyse préliminaire, nous avons aussi observé un cas particulier avec une application parallèle OpenMP. Cette application simule un problème de déséquilibre de travail entre les fils d'exécution. Par défaut, OpenMP utilise des verrous actifs pour la barrière de synchronisation. L'attente dans les verrous actifs dans les applications n'est pas visible du point de vue du système d'exploitation. En passant la variable d'environnement OMP_WAIT_POLICY=PASSIVE, alors le blocage dans un appel système futex se produit et est donc devient visible depuis le système d'exploitation. La figure 6 montre la différence entre l'attente active et l'attente passive de l'application OpenMP tel que vu par une trace noyau, pour la même portion d'exécution du programme. Ainsi, il faut que les appels systèmes soient utilisés pour que le comportement d'attente soit observé depuis le système d'exploitation, sans quoi cette analyse n'est pas possible. Habituellement, les verrous actifs sont réservés aux situations d'attente de courte durée, donc il devrait y avoir un faible impact sur la précision des résultats. De la même manière, les fils d'exécution en espace utilisateur et la communication en mémoire partagée risquent de poser problème, car ils peuvent être invisibles du point de vue du système d'exploitation. Ces éléments doivent être tenu en considération lors de l'évaluation du système.

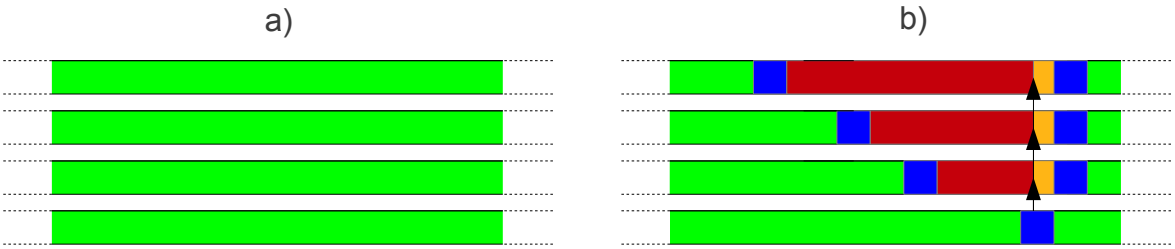


Illustration 6: Déséquilibre d'une application OpenMP

L'analyse préliminaire nous indique que notre approche a un potentiel d'être générique pour un grand nombre d'applications et de permettre de retrouver le chemin critique réel d'une application non limitée par l'utilisation du processeur. Les limitations concernent les primitives de synchronisation, de fils d'exécution et de communication qui ne serait pas visible depuis l'espace noyau.

2.2 Extraction du graphe d'exécution

2.2.1 Sémantique

La première étape est de développer la sémantique complète de la construction du graphe d'exécution sur un ordinateur local. Le fait de développer l'algorithme en local en premier lieu permet de simplifier l'analyse en ayant une horloge globale. Il est prévu que l'algorithme développé s'exécute hors ligne en premier lieu. L'accent est mis sur la précision et l'exactitude du graphe. Cet algorithme servira de base comparative pour tout autre algorithme développé ultérieurement.

Le prototype actuel extrait le graphe pour des processus parents et enfants. Cet algorithme doit être complété pour gérer tous les cas de communication et de synchronisation inter-processus. La lecture et l'écriture dans des fichiers, tubes anonymes et nommés, ainsi que les sockets Unix sont à étudier. Ces mécanismes ont un fonctionnement similaire, donc leur traitement devrait être analogue. À ceux-ci s'ajoute les verrous implémentés par une attente passive, visibles depuis l'espace noyau.

L'attente pour une entrée sortie, comme un accès disque, doit être identifiée correctement. Pour réaliser cette analyse, les évènements de faute de page doivent être considérés. Il doit être possible de relier cette attente à la requête du disque, car celui-ci est sur le chemin critique si l'application est bloquée en attendant la terminaison de la requête. Cette situation se produit pour les pages absentes en cache et les fautes de page majeures. Cette analyse nécessite de suivre la requête du disque, de l'envoi de la requête DMA jusqu'à l'émission de l'IRQ, suivi du SoftIRQ puis du signal de réveil.

Dans une autre catégorie se trouvent les sockets IP locaux. Ce mécanisme de communication se distingue par le fait qu'il est asynchrone. Du point de vue de l'émetteur, l'écriture dans un socket se traduit par l'émission de paquets en arrière-plan. Du côté du récepteur, le paquet reçu génère un IRQ, puis un SoftIRQ qui cause un signal de réveil dans l'application si celle-ci était bloquée en attente du message. Le défi consiste donc à suivre le chemin de communication entre l'appel système de transmission, son cheminement à travers la couche réseau, son envoi et sa réception par le périphérique et finalement jusqu'à la lecture du message par le récepteur. Le chemin complet incluant l'envoi et la réception de chaque message est certainement très précis et exhaustif, mais requiert en contrepartie de tracer tous les paquets émis et reçus. Une alternative serait d'approximer le chemin de communication en observant les blocages et les réveils des appels systèmes seulement. Les deux méthodes seraient à comparer par rapport à leur surcout et leur précision.

Le graphe d'exécution peut servir à des analyses subséquentes. L'arbre de dépendance entre les processus d'un système peut être déterminé automatiquement en parcourant le graphe d'exécution. La figure 7 présente un exemple d'interaction d'un navigateur web lors de l'accès à une page. Le résultat est une représentation statique de tous les systèmes impliqués pendant le traitement.

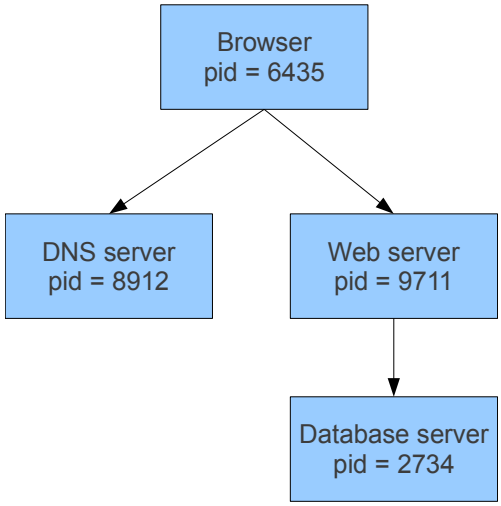


Illustration 7: Arbre de dépendance pour une requête d'un navigateur web

L'accent sera mis sur l'analyse du chemin critique du graphe d'exécution, discuté à la section 2.3.

2.2.2 Synchronisation

L'algorithme de calcul de graphe d'exécution doit être étendu pour supporter plusieurs traces enregistrées simultanément sur plusieurs ordinateurs aux horloges indépendantes. La synchronisation temporelle des traces doit être considérée. Pour que le graphe soit cohérent, l'annotation du temps sur les arcs doit être positif. Ceci exige un ordre partiel sur les évènements d'échanges de messages. La synchronisation à l'aide des évènements des paquets réseau échangés a été étudiée antérieurement [8], [9]. La précision mesurée expérimentalement sur un réseau Ethernet gigabit avec l'algorithme de synchronisation Convex-Hull est de 30µs. Aucune inversion de message n'a été observée lors de l'expérience. L'utilisation de cet algorithme est envisagée pour la synchronisation des traces. La corrélation des traces MPI et noyau indique la faisabilité de cette approche [7].

2.2.3 Agrégation

Du point de vue de l'agrégation de la trace distribuée, nous nous limiterons à une architecture de type client-serveur. Nous savons que cette architecture impliquant un point central limite la mise à l'échelle de l'analyse, mais elle est suffisante pour démontrer l'intérêt de notre approche. Si l'infrastructure d'agrégation et de contrôle en arbre devient disponible en cours du projet, nous pourrions envisager de contribuer un module améliorant la mise à l'échelle de l'analyse.

2.2.4 Instrumentation

Des points de trace noyau supplémentaires sont probablement requis pour réaliser cette analyse. Le choix de l'emplacement des points de trace et de l'information à extraire doit être déterminé en fonction des besoins de l'analyse, mais aussi en considérant l'acceptabilité des changements en vue de l'inclusion dans le noyau Linux. Des modifications aux algorithmes du noyau pour satisfaire les besoins de traçage sont peu susceptibles d'être adoptées. Cette contrainte indique qu'il faut trouver des identifiants in situ pour réaliser la

relation transitive des évènements. La disponibilité de tels identifiants cohérents et accessibles à plusieurs niveaux du noyau est incertaine.

2.2.5 Traçage juste à temps

Notre objectif est de tracer une application distribuée tout en impliquant un surcout minimal. Le traçage noyau enregistre habituellement tous les évènements activés globalement. Ceci est souhaitable pour obtenir des mesures de performances pour le système, mais seulement un sous-ensemble d'évènements est requis pour l'extraction du graphe d'exécution. Nous proposons d'étudier le traçage juste à temps pour cerner les évènements pertinents à notre analyse et diminuer drastiquement le surcout.

Le traçage juste à temps vise à tracer seulement les processus impliqués dans un graphe d'exécution à partir d'un processus racine. La fonctionnalité requise du point de vue du traceur est l'activation du traçage pour un ensemble de processus. Aux points de divergence du flot de contrôle, le processus en relation est ajouté à l'ensemble des processus tracé. La désactivation du traçage pour un processus peut se faire lorsqu'une condition est satisfaite, comme la fermeture du descripteur de fichier utilisé pour la communication ou la fin du processus.

La propagation de l'activation du traçage doit aussi traverser les frontières d'une machine. Il est proposé d'étudier la propagation en utilisant des options du protocole TCP. Le processus dont le traçage est activé définit une option TCP qui indique au récepteur de déclencher le traçage pour le processus auquel est destiné le paquet. La désactivation du traçage pourrait se produire lors de la réception d'un paquet pour laquelle l'option est absente.

L'activation du traçage doit être optimisée pour avoir un impact minimal sur la latence du processus. Habituellement, les points de trace sont activés avant de démarrer le traçage et l'impact de l'activation dynamique n'a pas été étudié.

2.2.6 Échantillonnage

L'échantillonnage est une autre technique proposée pour diminuer le surcout du traçage. L'échantillonnage aléatoire des évènements n'est pas adapté à notre analyse, car la perte

d'un seul évènement peut compromettre l'extraction du graphe. Plutôt que de faire l'échantillonnage sur les évènements, il est envisagé de l'appliquer au niveau d'un graphe d'exécution complet. Sur un système chargé, un déblocage aléatoire d'un processus en observation activerait le traçage. Le traçage serait propagé aux processus reliés tel que discuté précédemment. La requête serait suivie pendant toute la durée de son traitement, mais uniquement pour celle-ci.

2.2.7 Traçage spéculatif

Le traçage spéculatif est disponible avec Dtrace [49]. Ceci permet d'enregistrer en continu des évènements en mémoire, puis de déclencher leur écriture sur disque si une condition est satisfaite. Contrairement à l'activation dynamique, les points de trace sont toujours activés, ce qui implique un surcout constant, mais réduit la taille de la trace sur disque. L'utilisation du traçage spéculatif pourrait être utilisé pour n'enregistrer que les appels systèmes bloquant, les fautes de pages majeures et les évènements de réveil correspondants. Les appels systèmes et les fautes de page seraient tous tracés, mais enregistrés seulement si un changement de contexte lié à une attente passive survient.

2.2.8 Traitement en ligne

Nous désirons étendre les algorithmes et les outils pour qu'ils fonctionnent en ligne. Le mode en ligne dont il est question produit l'analyse de façon incrémentale à mesure que les évènements sont disponibles. Le traitement en ligne requiert l'adaptation de plusieurs algorithmes : la synchronisation temporelle, la construction du graphe et le calcul du chemin critique (discuté à la section 2.3) doivent tous se faire de manière incrémentale. Des avancées récentes à paraitre pour la synchronisation de traces incrémentale en ligne seront utiles pour la réalisation du système.

2.3 Calcul du chemin critique

Le calcul du chemin critique prend en entrée le graphe d'exécution et retourne le sous-graphe qui limite la durée d'exécution. L'algorithme de parcours arrière du graphe a été présenté brièvement à la section 2.1. Cet algorithme a l'avantage d'être déterministe et

d'avoir un temps d'exécution proportionnel à la longueur du chemin critique. L'algorithme peut commencer lorsque le graphe complet est disponible.

Nous proposons d'étudier aussi le parcours vers l'avant du graphe et de comparer les méthodes. L'avantage est de calculer le chemin critique lors de la construction de graphe et de la lecture de la trace, ce qui est adapté au traitement en ligne. Cependant, cette recherche en largeur nécessite d'explorer le graphe complet.

Il est nécessaire de lire tous les évènements de la trace pour retrouver l'état de chaque processus dans le temps, ce qui implique que le temps de traitement est proportionnel à la taille de la trace, ce qui est le meilleur temps sans prétraitement. Nous souhaitons étudier l'utilisation d'index pour réduire la complexité algorithmique d'extraction du graphe. Ces index sont avantageux, car le cout de l'indexation peut être réparti sur plusieurs analyses subséquentes. L'indexation peut se faire en ligne et donc diminuer le délai de l'analyse perçue par l'utilisateur.

Par exemple, l'indexation de certains types d'évènements de la trace permettrait de lire uniquement les portions qui sont utiles à la construction du graphe en sautant les évènements qui ne sont pas pertinents. L'accélération produite serait en fonction de la proportion de la trace non lue.

L'index d'intervalle sur disque [15] permet de retrouver l'état complet du système à un moment précis en temps logarithmique. Le temps de construction de l'index est proportionnel à la taille de la trace, mais constant en mémoire. L'index a été testé avec succès pour des traces jusqu'à 1To. Cet index permet un déplacement suivant les changements d'état d'un attribut. Par exemple, un attribut peut être l'état d'un processus. Les intervalles ont un début et une durée, ce qui permet d'itérer directement sur les changements d'état plutôt que sur les évènements de la trace. L'avantage pour le calcul du graphe d'exécution est de parcourir les états bloqués des processus, car ceux-ci indiquent le changement du flot de contrôle. L'index d'intervalle a donc le potentiel de réduire la complexité algorithmique de manière proportionnelle au nombre de blocages plutôt qu'au nombre d'évènements. Le gain de performance attendu est en fonction du ratio

d'évènements par rapport au nombre d'intervalles. Ce ratio dépend des caractéristiques de l'application observée et doit être vérifié expérimentalement.

2.4 Garanties de traçage

La méthode précise proposée nécessite qu'aucun évènement ne soit perdu, sans quoi les graphes pourraient être erronés et rendre toute l'analyse inutile. La génération d'évènements en mémoire excédant la bande passante du disque ou du réseau cause l'écrasement des plus anciens évènements dans le tampon circulaire. Ce design fait en sorte de mettre une borne supérieure à la perturbation du système.

Le problème de perte d'évènements peut se traiter sous deux angles différents. La première consiste à développer un algorithme qui possède une certaine tolérance aux pertes. Cette solution a été étudiée dans PreciseTracer [2]. La tolérance aux pertes contribue à rendre l'algorithme plus robuste et pourrait produire des résultats approximatifs pour un faible taux de perte. Cette tolérance est en général désirable et doit être évaluée. L'autre solution consiste à sacrifier les performances pour la précision en forçant le système à enregistrer tous les évènements générés en toutes circonstances.

Garantir qu'aucun évènement ne puisse être perdu est non trivial dans le cas du traçage noyau, car le système d'exploitation génère des évènements et ne peut pas être arrêté pendant l'écriture sur disque (ou l'envoi par réseau) de la trace. Aussi, la production d'évènements est gouvernée par un processus stochastique en rafale. Les pistes de solutions suivantes sont envisagées pour l'étude.

- Allocation dynamique de la taille des tampons : simplifie la procédure pour déterminer la taille appropriée des tampons requis pour gérer les rafales d'évènements. Si le débit de production d'évènements moyen en régime permanent excède la bande passante du disque, cette méthode n'est pas efficace pour ne garantir aucune perte.
- Désactivation dynamique d'évènements : en désactivant des évènements moins prioritaires lorsque la capacité des tampons dépasse un certain seuil, le débit de

production serait diminué. Cette méthode ne garantit pas l'enregistrement de tous les évènements, mais priorise les pertes.

- Prioriser le démon de synchronisation : si le démon est plus prioritaire et qu'il ne cède pas les processeurs, alors cela a pour effet de limiter la production d'évènements par les autres processus sur le système. Les évènements liés à l'ordonnancement et aux interruptions seront tout de même émis. Cette méthode pourrait garantir qu'aucun évènement ne soit perdu si la bande passante est supérieure au débit de génération d'évènements liés à la synchronisation elle-même.
- Utiliser le mode d'enregistrement différé : dans ce mode, les évènements sont enregistrés en continu dans le tampon circulaire jusqu'à ce qu'une condition soit satisfaite. À ce moment, le traçage est arrêté et les tampons sont écrits sur disque. Cette technique garantit que les derniers évènements seront écrits sans perte. Cependant, la durée de la fenêtre enregistrée varie en fonction du débit moyen des évènements produits et de la taille des tampons.

Nous envisageons de développer des euristiques combinant différentes approches, de manière à augmenter la robustesse. Un modèle analytique du système permettrait de simuler la performance de différentes euristiques rapidement, puis d'implémenter la solution qui présente les meilleures caractéristiques et de l'évaluer dans des conditions extrêmes.

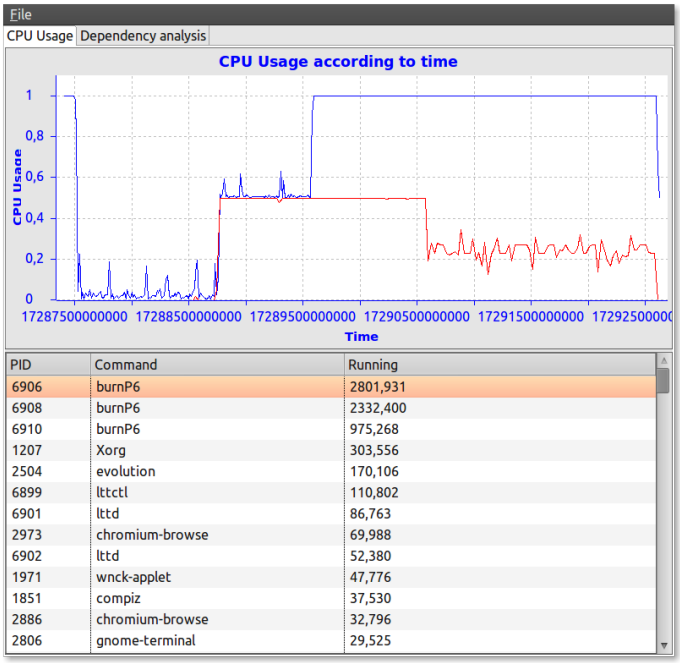
2.5 Métriques de performance

Les métriques du système sont requises pour déterminer le cout d'une exécution distribuée. Nous avons identifié l'instrumentation du noyau utile pour extraire les métriques du système [52]. Les métriques sont l'utilisation des processeurs, de la mémoire, des disques et du réseau. Nous avons développé un prototype pour montrer la faisabilité. Une capture d'écran de ce prototype est montrée à la figure 8. L'exemple montre le démarrage de trois processus exécutant une boucle de calcul sur une machine à deux processeurs. La contention apparait après le démarrage du troisième processus. Le taux d'utilisation du processeur dans le temps calculé à partir des évènements d'ordonnancement. Contrairement à l'interface /proc pour lire les valeurs moyennes d'utilisation du processeur périodiquement, cette méthode permet

de détecter des processus d'une très courte durée. Nous notons aussi une meilleure précision, car la période sur laquelle la moyenne est calculée est personnalisable. En définissant une période nulle, alors l'onde carrée représentant les changements de contextes serait retrouvée à partir d'une trace noyau, une précision qui n'est pas possible d'atteindre par scrutation.

En plus de ces métriques, il serait possible de jumeler la valeur des compteurs de performance du processeur à la trace noyau. Cette fonctionnalité faciliterait la corrélation de l'information entre les couches du système, tout en bénéficiant du faible impact de performance encourue par les compteurs de performance.

Nous souhaitons étendre les algorithmes de calcul des métriques pour accumuler toutes les ressources utilisées sur le graphe d'exécution. Les arcs du graphe seraient annotés avec les ressources utilisées pendant cet intervalle. Le cout d'une exécution pourrait donc être calculé avec une très grande précision.



PID	Command	Running
6906	burnP6	2801,931
6908	burnP6	2332,400
6910	burnP6	975,268
1207	Xorg	303,556
2504	evolution	170,106
6899	lttctl	110,802
6901	lttd	86,763
2973	chromium-browse	69,988
6902	lttd	52,380
1971	wnck-applet	47,776
1851	compiz	37,530
2886	chromium-browse	32,796
2806	gnome-terminal	29,525

Illustration 8: Calcul de taux d'utilisation du processeur depuis une trace noyau

2.6 Évaluation

L'infrastructure de traçage LTTng et le format de trace CTF seront utilisés pour le développement expérimental. Le code source ouvert permet de modifier le logiciel pour implanter nos algorithmes et réaliser nos mesures.

Deux aspects sont considérés pour l'évaluation des algorithmes d'extraction de graphes développés, soit la précision et le surcout. Notre stratégie d'évaluation est semblable à vPath [37]. Les techniques doivent être adaptés au contexte des traces noyau.

Pour évaluer la précision des algorithmes, il faut des programmes ayant un comportement connu. À cette fin, nous avons démarré le projet workload-kit, qui contient une série de programmes produisant une charge connue sur le système. Par exemple, nous avons intégré un algorithme utilisant une étape de calibration pour produire une charge de calcul prédéterminée approximative, peu importe la fréquence du processeur. Ces utilitaires seront étendus pour simuler toutes les situations de base à observer. Un utilitaire sera développé pour générer des charges distribuées arbitraires. Cet utilitaire enregistrera son propre graphe d'exécution, qui servira de base pour comparer le graphe retrouvé depuis la trace noyau.

Comme nos algorithmes dépendent des caractéristiques du réseau, nous allons étudier l'extraction du graphe en simulant une latence élevée. Ce test devrait rendre visibles tous les blocages qui pourraient survenir dans une application et permettre d'étudier le comportement lorsque les tampons du noyau sont pleins.

Les outils développés seront validés sur des applications réelles, pas seulement sur les programmes de tests. RUBiS [53] est un service web de transaction en Java répliquant les fonctionnalités du site d'enchères eBay. Il est possible de simuler la charge d'un nombre arbitraire de clients simultanés. Nous avons l'intention de faire le traçage du client et de l'infrastructure web pour montrer le fonctionnement approprié de notre méthode. Les résultats seront vérifiés par inspection, tandis que le surcout sera mesuré expérimentalement selon le nombre de clients simultanés.

La robustesse du traceur à ne perdre aucun évènements sera étudiée en produisant une charge extrême sur le système. Nous allons déterminer expérimentalement la charge

combinée produisant le débit d'évènements le plus élevé. Le traçage sera réalisé dans ces conditions, puis le nombre d'évènements perdus sera observé. Nous allons aussi observer le comportement en faisant varier le débit du périphérique servant à la sauvegarde de la trace. Le modèle analytique sera validé à l'aide des observations empiriques.

Pour réaliser l'évaluation du traçage en ligne, nous utiliserons la grappe de calcul de 32 processeurs du laboratoire DORSAL. La précision des résultats obtenus en ligne sera comparée à l'algorithme hors ligne. Le surcout sera évalué en fonction de la fréquence de l'analyseur.

Nous allons utiliser les outils développés pour résoudre des problèmes de performances que rencontrent les partenaires industriels. Nous souhaitons montrer l'utilité des outils dans des situations réelles, pour lesquelles ils ont permis d'améliorer la résolution de problèmes de performance complexes sur des systèmes distribués.

2.7 Travaux connexes

Dans le cadre du projet, nous prévoyons faire des développements qui ne sont pas associés à une contribution scientifique, mais qui sont néanmoins nécessaires. Par exemple, nous désirons développer une interface graphique pour afficher le graphe d'exécution et son chemin critique. Cette interface graphique sera développée en Java sous forme de plugin Eclipse. Cette interface sera publiée en tant qu'application autonome dont le nom est LTTng Studio.

3 Planification

Le déroulement du doctorat est prévue sur 3,5 ans. Pendant cette période, la publication de trois articles de journaux est planifiée. Cette section détaille les contributions attendues et l'échéancier de réalisation.

3.1 Contributions

Les programmes issus de la recherche seront publiés dans le but d'obtenir des commentaires de nos partenaires industriels. Les contributions logicielles visent les composants suivants:

- Noyau Linux : Instrumentation additionnelle.
- LTTng : Traçage juste à temps, garantie de traçage.
- Workload-kit : Programmes de test.
- LTTng Studio : Algorithmes d'analyse et de visualisation.

La première étape est de réaliser l'extraction du graphe d'exécution hors ligne, qui est préalable à l'extraction en ligne et au calcul du chemin critique. Les contributions concernant les optimisations et la robustesse sont planifiées en dernier, car ils visent à améliorer les résultats obtenus précédemment. La contribution précise du dernier article sera ajustée selon l'évolution du projet et les opportunités de recherche en cours. Les alternatives prévues sont toutefois présentées.

Le reste de la section détaille la portée des articles prévus. Ensuite, les journaux ciblés pour la publication sont listés. En dernier lieu, les conférences pertinentes pour la dissémination des résultats sont présentées.

3.1.1 Articles scientifiques

Voici la liste des articles prévus dans le cadre de la recherche.

Article 1 : Dependency analysis from kernel trace

Résumé : With the increasing complexity of IT systems in the cloud, new tools are required to observe precisely the runtime execution of applications for debugging or profiling purposes. We propose a new method to observe the runtime behavior of distributed applications from a kernel trace. Tools work with any recent Linux kernel, without modifications to applications nor libraries, and work accros machine boundaries. The method is based on blockings that can occur inside system calls to infer change in the control flow. The method extracts execution graph with low runtime overhead and compute the cost of the whole execution. We present LTTng Studio, a tool to visualize the distributed execution of an application. Our results are presented in terms of accuracy, runtime overhead and algorithmic complexity. We used LTTng Studio to observe a set of basic

distributed system primitives and the end-to-end performance of the RUBiS three tier auction web site.

Article 2 : Online critical path extraction of distributed applications from kernel trace

Résumé : Characterizing with precision the time spent in each component of a heterogenous distributed application, in response to a request, is a challenging task. Yet, the response time is an important metric for user satisfaction. We present a novel technique to analyse the critical path of a distributed application at the operating system level. Other techniques were able to compute the critical path in terms of CPU usage. While it is effective for most scientific computations, we show that fails for mixed CPU and I/O workloads of typical web transaction systems. We address this limitation by following control flow changes uppon blocking and wakeup of distributed processes. We compare offline and online algorithms to compute the critical path. We evaluate the capability of the tool to recover critical path of parallel applications and a three tier transaction system, in terms of precision and runtime overhead.

Article 3 a) : Just in time kernel tracing

Résumé : We demonstrated previously a method based on kernel tracing for critical path recovery of distributed applications. We propose just in time tracing to improve drastically the scalability of the analysis. This technique records the minimal set of events required to recover the execution graph. It works by activating the instrumentation by following the control flow of the application at runtime, across machine boundaries. To increase scalability even more, we propose to use a sampling technique that records only a subset of all executions. The sampling can be adjusted to provide desired maximum overhead. We use a transaction web site representative of commercial workloads to compare accuracy and overhead between complete and just in time tracing.

Article 3 b) : Preventing event loss of operating system tracer

Résumé : Kernel tracing is a precise and rich source of runtime information. Precise analysis performed on kernel events are subject to fail in case of event loss. For example, if a scheduling event is lost on a CPU, then following events are accounted to the wrong

process. Event loss occurs when circular buffers are full and generated events overrides ones that are not already consumed. This situation has a higher probability to occur under abnormal load, exactly when tracing would be usefull to understand the problem. In the traditionnal producer and consumer problem, the producer blocks until some buffer space is freed. In the case of kernel tracing, blocking the kernel to flush buffers would result in an immediate deadlock of the system. Increasing buffer size can handle event bursts, but the average event rate must always be lower than the disk or network bandwith. The analysis is complexified by the presence of a positive feedback loop in the system, in which the consumer activity is recorded and generates events, that may lead to instability. We developed a model of event production and consumption of kernel tracing to study this problem. We describe the model validation emperically according to a set of workloads. We evaluate the effectiveness and cost of various heuristics in terms of their ability to prevent event loss.

3.1.2 Revues et journaux

Nous avons ciblé trois journaux avec revue par les pairs pour la publication des travaux. Les facteurs d'impacts ont été obtenus depuis la base de donnée ISI Web Of Knowledge[1].

ACM Transactions on Computer Systems (TOCS)

Description : Ce journal se dédie à la recherche sur les ordinateurs et accepte les articles portant sur les systèmes d'exploitation, les modèles de performance et l'analyse de performance, ce qui correspond à la portée de l'ensemble de nos contributions. Facteur d'impact: 1,889

IEEE Transactions on Computers

Description : Publication mensuelle portant entre autres sur les systèmes d'exploitation et l'analyse de performance. Fondé en 1953, il était en 2004 le 16^{e} journal le plus cité dans le domaine du génie électrique et électronique. Les sujets couverts sont en lien direct avec la nature de nos contributions. Facteur d'impact: 1,608

1 ISI Web of Knowledge – Journal Citation Reports® http://admin-apps.webofknowledge.com

IEEE Transactions on Parallel and Distributed Systems

Description : Publication mensuelle sur les mesures de performances sur des grappes d'ordinateurs, les systèmes d'exploitation, les modèles de performances et la simulation de systèmes multiprocesseurs. L'aspect distribué de notre méthode et la capacité d'étudier des applications parallèles sont pertinents pour ce journal. Facteur d'impact : 1,575

3.1.3 Conférences

Voici une liste de cinq conférences pertinentes pour la dissémination de nos recherches.

ACM SIGOPS EuroSys

Description : EuroSys est le chapitre européen du groupe d'intérêt ACM SIGOPS portant entre autres sur les systèmes d'exploitations, les architectures multiprocesseurs et multiprogrammées et la modélisation des ordinateurs. La conférence EuroSys a lieu annuellement au printemps.

ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)

Description : Les sujets qui sont pertinents à notre recherche sont la vérification et le test des systèmes distribués, les architectures multiprocesseurs et la programmation parallèle. La conférence PODC se déroule annuellement à l'été.

ACM Symposium on Operating Systems Principles (SOSP)

Description : La conférence couvre les sujets relatifs aux systèmes d'exploitation et les systèmes distribués. Il est mentionné que les articles qui se distinguent peuvent être référés au journal ACM Transactions on Computer Systems en vue d'une possible publication. La conférence a lieu annuellement à l'automne.

Runtime Verification

Description : La conférence Runtime Verification traite de l'observation en fonctionnement des systèmes et des analyses associées. Les aspects pertinents à notre recherche sont l'instrumentation des programmes, le traçage et la visualisation de l'exécution des programmes. La conférence a lieu annuellement à l'automne.

USENIX Symposium on Operating Systems Design and Implementation (OSDI)

Description : Conférence organisé par The Advanced Computing Systems Association sur le même thème que ACM SOPS. La conférence a lieu annuellement à l'automne.

IEEE International Symposium on Performance Analysis of Systems and Software

Description : La conférence spécifie rechercher des articles dans le domaine du traçage et de l'analyse de performance, en particulier pour des systèmes distribués. La conférence a lieu au printemps.

3.2 Échéancier

L'échéancier du projet est montré à la figure 9. Le projet a commencé en janvier 2011. Pendant cette période, la revue de littérature et le prototypage ont été réalisés. Ces travaux ont mené à la rédaction de cette proposition de recherche. Tous les cours académiques requis seront complétés en avril 2012.

Un stage chez Ericsson au siège social en suède dans le groupe Component Based Architecture (CBA) est planifié pour la session d'été 2012. Les activités planifiées seront conduites sur place.

Les principales étapes de la réalisation de chaque article sont documentées. En ce qui concerne la dernière contribution, seule une des alternatives est présentée. En cas de changement pour le dernier sujet, son échéance demeure la même.

TPÉ	Nom	Travail
1	**Proposal Francis Giraldeau**	**891j**
1.1	Preliminary analysis	150j
1.2	Litterature review	150j
1.3	Research method course	14j
1.4	Comprehensive exam	1j
1.5	Proposal writing	28j
1.6	Oral exam	1j
1.7	**Execution graph recovery**	**175j**
1.7.1	Establish execution graph semantic	20j
1.7.2	Implement kernel instrumentation	20j
1.7.3	Develop workload-kit programs	20j
1.7.4	Port trace sync to Java	20j
1.7.5	Develop offline extraction algorithm	20j
1.7.6	Conference paper: early results	14j
1.7.7	Complete system metrics recovery	20j
1.7.8	Develop graph visualisation	20j
1.7.9	Gather results	10j
1.7.10	Write paper	10j
1.7.11	Submit paper	1j
1.8	Conference early results	N/D
1.9	Conference	N/D
1.10	**Critical path computation**	**141j**
1.10.1	Study reverse traversal	10j
1.10.2	Study forward traversal	10j
1.10.3	Indexed critical path computation	20j
1.10.4	Online synchronisation	20j
1.10.5	Online graph recovery	20j
1.10.6	Online critical path	20j
1.10.7	Network experiments	20j
1.10.8	Gather results	10j
1.10.9	Write paper	10j
1.10.10	Submit paper	1j
1.11	Conference	N/D
1.12	**Just in time tracing**	**185j**
1.12.1	Study process control flow	40j
1.12.2	Develop/adjust filtering capabilities	40j
1.12.3	Implement speculative tracing	40j
1.12.4	Implements kernel instrumentation	40j
1.12.5	Gather results	10j
1.12.6	Write paper	10j
1.12.7	Submit paper	5j
1.13	Write thesis	45j
1.14	Thesis defense	1j

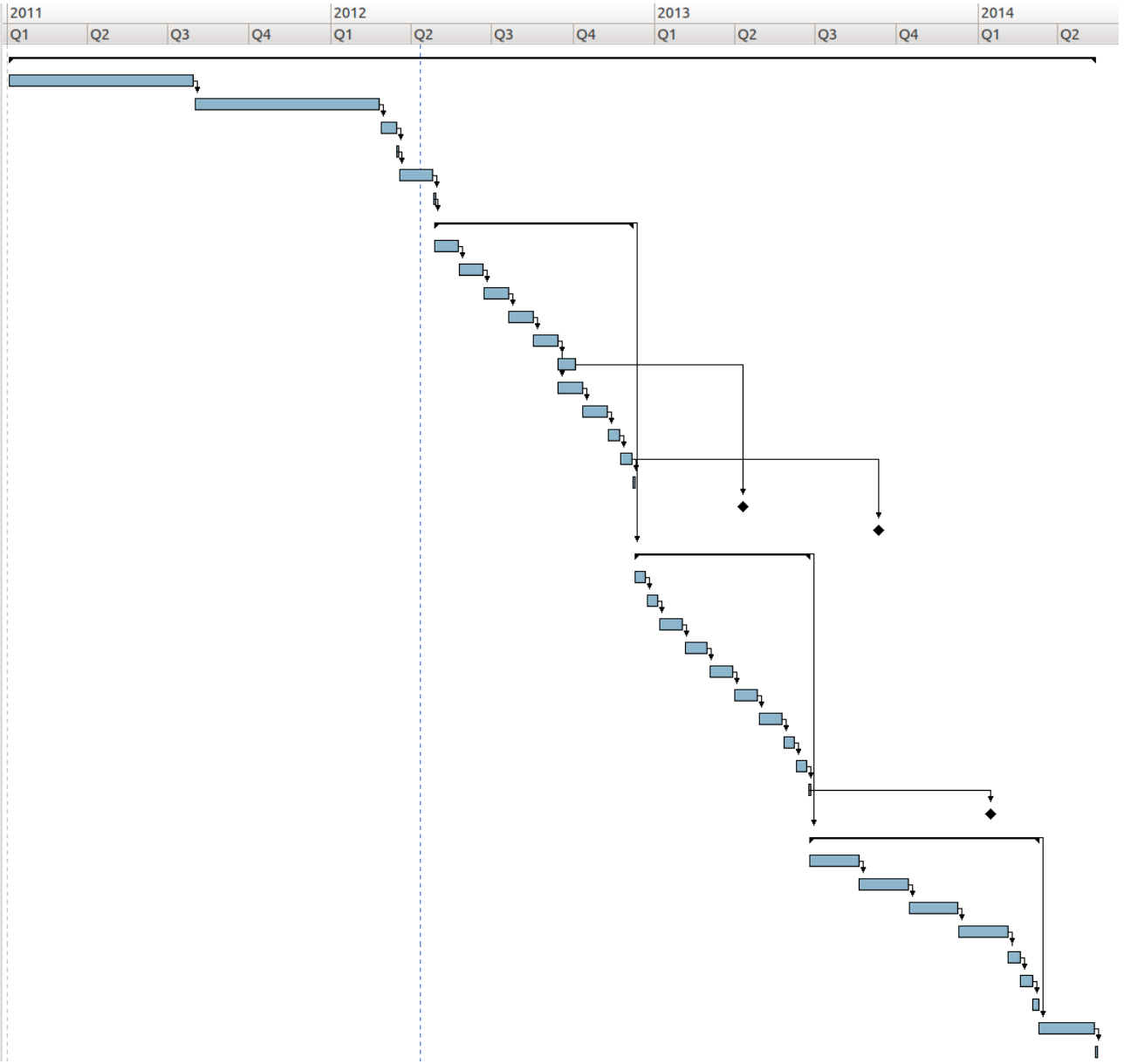


Illustration 9: Diagramme de Gantt de la recherche

4 Travaux futurs

Certains liens entre processus ne sont pas visibles depuis l'espace noyau, notamment les verrous actifs, la communication par mémoire partagée et les fils d'exécution en mode utilisateur. Ces limitations affectent la précision de l'analyse pour certaines applications. L'instrumentation facile serait d'ajouter des points de trace dans les librairies en espace utilisateur. Cependant, le défi est plus grand pour observer ces primitives de programmation d'une manière transparente, sans modification aux librairies, et implique des compromis du point de vue de la performance ou de la précision. Dans le cas de la mémoire partagée, elle peut être suivie en protégeant les pages de mémoire virtuelles, cependant le traitement de l'interruption à chaque accès est très couteux, il serait intéressant de rechercher une méthode plus efficace. En ce qui concerne les fils d'exécution en espace utilisateur, il serait pertinent de vérifier la possibilité d'utiliser le registre du pointeur de base pour détecter les changements de fils d'exécution en espace utilisateur. L'efficacité de cette technique dépend des conventions utilisées par les ce type de librairies et de l'efficacité à détecter le changement du pointeur de base. Finalement, les verrous actifs sont peut-être identifiables en établissant leur profil d'exécution obtenu par des compteurs de performance lors d'une contention, mais il s'agit d'une approche statistique pour laquelle il existe une incertitude.

Dans un autre ordre d'idée, l'établissement d'un modèle LQN du point de vue du système d'exploitation pourrait déterminer de manière générique les goulots dans un système distribué. Le modèle serait alors alimenté par les évènements d'une trace noyau et fournir en continu les paramètres de performance à jour du système pour sa surveillance, ou utilisé dans le cadre d'un plan d'assurance qualité.

En dernier lieu, nous avons planifié démontrer notre système avec Linux en acceptant comme hypothèse que ce système d'exploitation est une constante. Dans la réalité, il existe une diversité de système d'exploitation en usage. Or, le traçage noyau étant disponible sous d'autres systèmes d'exploitation, il serait pertinent d'étudier la compatibilité de la sémantique des évènements et de l'abstraire, dans le but de réaliser l'analyse de l'exécution distribuée indépendante du système d'exploitation. La difficulté de réalisation concerne

l'impédance probable entre l'instrumentation des différents systèmes d'exploitation, car il n'existe pas à notre connaissance de standard dans ce domaine.

5 Conclusion

Le traçage noyau fournit des informations de premier plan pour comprendre le comportement d'un système en exécution. Cette méthode est peu intrusive, a un faible surcout et offre une vue globale du système. Cette proposition de recherche vise à continuer les efforts du laboratoire DORSAL dans le domaine du traçage noyau. D'une part, en développant les algorithmes et les outils pour l'extraction du graphe d'exécution d'une application distribuée, puis en calculant son chemin critique. Nous pensons que cette analyse originale aidera ses utilisateurs à comprendre comment le temps d'exécution se répartie entre les composants du système, ce qui serait impraticable à faire de manière générique avec d'autres méthodes, ou même manuellement sur des traces noyau obtenues sur des systèmes complexes. D'autre part, la proposition concernant le traçage juste à temps a le potentiel de réduire l'ordre du surcout de l'analyse, rendant possible son activation en continu sur des systèmes en production. Ces analyses pourraient ainsi changer la manière dont on voit les systèmes distribués à l'avenir et rendre le traçage noyau un outil de travail incontournable en génie informatique.

6 Références

[1] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, "Field studies of computer system administrators: analysis of system management tools and practices," in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, 2004, pp. 388–395.

[2] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, and Z. Zhang, "Precise, scalable, and online request tracing for multi-tier services of black boxes," *IEEE Transactions on Parallel and Distributed Systems*, no. 99, p. 1–1, 2010.

[3] M. R. Dagenais, K. Yaghmour, C. Levert, and M. Pourzandi, "Software Performance Analysis," *Arxiv preprint cs/0507073*, 2005.

[4] M. Desnoyers, "Low-impact operating system tracing," Thesis, École Polytechnique de Montréal, Montréal, 2009.

[5] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *Proceedings of the Ottawa Linux Symposium*, 2006, vol. 2006.

[6] M. Desnoyers and M. Dagenais, "Low disturbance embedded system tracing with linux trace toolkit next generation," in *ELC (Embedded Linux Conference)*, 2006.

[7] B. Poirier, R. Roy, and M. Dagenais, "Unified Kernel and User Space Distributed Tracing for Message Passing Analysis," in *Proceedings of the First International Conference on Parallel, Distributed and Grid Computing for Engineering*, 2009, pp. 218–234.

[8] E. Clément and M. Dagenais, "Traces synchronization in distributed networks," *Journal of Computer Systems, Networks, and Communications*, p. 5, 2009.

[9] B. Poirier, R. Roy, and M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010.

[10] D. Toupin, "Using tracing to diagnose or monitor systems," *IEEE Software*, vol. 28, no. 1, pp. 87–91, 2011.

[11] A. S. Sendi, M. Jabbarifar, M. Shajari, and M. Dagenais, "FEMRA: Fuzzy Expert Model for Risk Assessment," presented at the Fifth International Conference on Internet Monitoring and Protection (ICIMP), 2010, pp. 48–53.

[12] G. Matni and M. Dagenais, "Automata-based approach for kernel trace analysis," in *Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2009, pp. 970–973.

[13] P. M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 77–87, 2010.

[14] M. Desnoyers and M. Dagenais, "OS tracing for hardware, driver and binary reverse engineering in Linux," *CodeBreakers Journal*, vol. 1, no. 2, 2006.

[15] A. Montplaisir-Goncaves, "Stockage sur disque pour accès rapide d'attributs avec intervalles de temps," Mémoire, École Polytechnique de Montréal, Montréal, 2011.

[16] E. Koskinen and J. Jannotti, "Borderpatrol: isolating events for black-box tracing," in *ACM SIGOPS Operating Systems Review*, 2008, vol. 42, pp. 191–203.

[17] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: black-box performance debugging for wide-area systems," in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 347–356.

[18] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2eprof: Automated end-to-end performance management for enterprise systems," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007, pp. 749–758.

[19] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma, "Constellation: automated discovery of service and host dependencies in networked systems," *TechReport, MSR-TR-2008-67*, 2008.

[20] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *ACM SIGOPS Operating Systems Review*, 2003, vol. 37, pp. 74–89.

[21] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," *Google Research*, 2010.

[22] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[23] M. Casas, R. Badia, and J. Labarta, "Automatic analysis of speedup of MPI applications," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 349–358.

[24] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of MPI resources*. Citeseer, 1996.

[25] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee, "Netlogger: A toolkit for distributed system performance analysis," in *Proceedings of 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000, pp. 267–273.

[26] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, "The NetLogger methodology for high performance distributed systems performance analysis," in *Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, 1998, pp. 260–267.

[27] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, "Stardust: tracking activity in a distributed storage system," in *ACM SIGMETRICS Performance Evaluation Review*, 2006, vol. 34, pp. 3–14.

[28] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[29] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan, "Request extraction in Magpie: events, schemas and temporal joins," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004, p. 17.

[30] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems," in *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, 2003, p. 15–15.

[31] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer, "Using runtime paths for

macroanalysis," in *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, 2003, p. 14–14.

[32] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski, "IPS-2: The second generation of a parallel program measurement system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206–217, 1990.

[33] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, 2005, vol. 40, pp. 190–200.

[34] S. Chiba, "Javassist: Java bytecode engineering made simple," *Java Developer's Journal*, vol. 9, no. 1, 2004.

[35] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood, "Understanding and visualizing full systems with data flow tomography," *ACM SIGPLAN Notices*, vol. 43, no. 3, pp. 211–221, 2008.

[36] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007, p. 20–20.

[37] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vPath: precise discovery of request processing paths from black-box observations of thread and network activities," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009, p. 19–19.

[38] A. Chanda, A. L. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, pp. 17–30.

[39] N. Froyd, J. Mellor-Crummey, and R. Fowler, "Low-overhead call path profiling of unmodified, optimized code," in *Proceedings of the 19th annual international conference on Supercomputing*, 2005, pp. 81–90.

[40] A. Mirgorodskiy and B. Miller, "Diagnosing distributed systems with self-propelled instrumentation," *Middleware 2008*, pp. 82–103, 2008.

[41] C. Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *8th International Conference on Distributed Computing Systems*, 1988, pp. 366–373.

[42] J. K. Hollingsworth, "An online computation of critical path profiling," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1996, pp. 11–20.

[43] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "Full-system critical path analysis," in *IEEE International Symposium on Performance Analysis of Systems and software (ISPASS)*, 2008, pp. 63–74.

[44] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2009.

[45] G. Franks, D. Lau, and C. Hrischuk, "Performance measurements and modeling of a java-based session initiation protocol (SIP) application server," in *Proceedings of International Symposium on Architecting Critical Systems (ISARCS)*, 2011, pp. 63–72.

[46] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th international workshop on Software and performance*, 2005, pp. 147–158.

[47] F. Ricciato and W. Fleischer, “Bottleneck detection via aggregate rate analysis: a real case in a 3G Network,” in *10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006, pp. 1–4.

[48] “Event Tracing for Windows.” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ff190903(v=vs.85).aspx. [Accessed: 30-Mar-2012].

[49] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004, p. 2–2.

[50] F. C. Eigler and R. Hat, “Problem solving with systemtap,” in *Proceedings of the Ottawa Linux Symposium*, 2006.

[51] “Red Hat Enterprise Linux Developer Guide.” [Online]. Available: http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Developer_Guide. [Accessed: 30-Mar-2012].

[52] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, “Recovering System Metrics from Kernel Trace,” in *Linux Symposium*, 2011, p. 109.

[53] “Rice University Bidding System.” [Online]. Available: http://rubis.ow2.org/. [Accessed: 12-Mar-2012].